**Relationale Datenbanken**

Entwicklung in den 1970er Jahren: Edgar F. Codd (IBM), Donald D. Chamberlin und Raymond F. Boyce.

Dies führte dann zum Datenbank-Standard SQL

SQL (Structured Query Language; strukturierte Abfragesprache) – ausgesprochen wie buchstabiert; manchmal auch wie „SEQUEL“ -> zum Aufbau und zur Verwaltung von relationalen Datenbanken.

Was ist eine relationale Datenbank?

* Eine relationale Datenbank ist hierarchisch aufgebaut: Die oberste Stufe stellen die Datenbanken selbst dar; in einer Datenbank befinden sich eine oder mehrere Tabellen und diese Tabelle(n) sind in Spalten und Zeilen organisiert; so können Daten mittels eines „koordinaten-artigen“ Systems angewählt werden.
* Koordinatensystem in Tabellen:

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | B | C |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

Um Daten in der dunkel schattierten Zelle auslesen zu können, kann man koordinaten-mäßig Spaltennamen und Zeilennummer angeben; hier also „B2“.

* Die Werte innerhalb einer Spalte müssen immer den gleichen Datentyp aufweisen (siehe später!).

**Merkmale von SQL**

* Relationale Datenbanksprache (manipuliert Tabellen und die darin enthaltenen Daten)
* Deskriptive Sprache: Man beschreibt die gesuchte Menge der Daten und nicht die Prozedur zum Finden der Daten! („very high level“ – Sprache)
* Sowohl interaktiv als auch in einem Programm/Skript benutzbar
* Entwickelt seit 1974 bei IBM; erster SQL-Standard 1986

SQL gliedert sich in:

* DDL (Datendefinitionssprache) zum Erstellen und Löschen von Datenbanken, Tabellen und Sichten (siehe später)
* DML (Datenmanipulationssprache) zum Einfügen, Löschen, Ändern und Abfragen von Elementen in Tabellen.

Grundoperationen in SQL

Bemerkung: SQL-Schlüsselwörter sind nicht case-sensitiv; d.h. man kann sie groß oder klein schreiben. Es hat sich aber eingebürgert, die Schlüsselwörter komplett GROSS zu schreiben, um einen besseren Überblick behalten zu können. Dieser Konvention wollen wir auch folgen.

Erstellen einer Datenbank

CREATE DATABASE Datenbankname;

Bemerkung: SQL-Statements werden mit einem Semikolon („;“) abgeschlossen. Ausnahmen dazu tauchen später noch auf; momentan gilt aber: Am Ende muss ein Semikolon stehen!

Bsp.:

CREATE DATABASE SQLInfo;

Achtung: Ob die Eigennamen, wie hier der Datenbankname, case-sensitiv ist, ist von SQL-Variante zu Variante unterschiedlich!  
Eigennamen wie DB- und Spaltennamen können teilweise Sonderzeichen und Leerzeichen enthalten; dann müssen diese Namen aber immer mit `einfachen Gänsefüßchen` eingeschlossen werden. Guter Ton ist es natürlich, dies, wenn möglich, zu vermeiden!

Löschen einer Datenbank

DROP DATABASE [IF [NOT] EXISTS] Datenbankname;

Erstellen einer Tabelle

CREATE TABLE Tabellenname  
 (Spaltenname Datentyp [NOT NULL …],

Spaltenname …

…

Spaltenname Datentyp [NOT NULL …]);

Es muss also mindestens für jede Spalte ein Name und ein Datentyp angegeben werden!

NOT NULL im obigen Code bedeutet: Das entsprechende Feld darf nicht leer sein; d.h. es muss gleich beim Erstellen eines Datensatzes dieses Feld auch gefüllt werden!

Datentypen gibt es in den verschiedenen SQL-Versionen viele; auch solche, die von Version zu Version nicht kompatibel sind (großes Problem in SQL!). Gängige Datentypen sind u.a.:

* (NULL kein/unbekannter Wert)
* BOOLEAN TRUE/FALSE (oder 1/0 etc.)
* INTEGER ganze Zahlen
* SMALLINT ganze Zahlen von -32767 bis +32767
* TINYINT ganze Zahlen von -128 bis 127
* MEDIUMINT ganze Zahlen von -8388608 bis 8388607
* BIGINT ganze Zahlen von ca. -9.2E18 bis 9.2E18
* FLOAT Dezimalzahlen von ca. -3.4E38 bis -1.1E-38
* DOUBLE Dezimalzahlen von ca. -1.7E308 bis -2.2E-308
* DECIMAL(p,s) Dezimalzahlen mit „p“ Stellen, davon „s“ Nachkommastellen

Bsp.: DECIMAL(6,2) kann Dezimalzahlen angeben von -9999.99 bis 9999.99

* CHAR(n) Zeichenkette der Länge „n“
* VARCHAR(n) Zeichenketten bis zur maximalen Länge „n“ (verbraucht nur die tatsächlich verwendeten Bytes).
* TINYTEXT Text bis 255 Zeichen
* TEXT Text bis 65535 Zeichen
* MEDIUMTEXT Text bis 16777215 Zeichen
* LONGTEXT Text bis ca. 4GB
* TINYBLOB Binäre Daten bis 255 Bytes
* BLOB Binäre Daten bis 65 kB
* MEDIUMBLOB Binäre Daten bis 16 MB
* LONGBLOB Binäre Daten bis 4 GB
* DATETIME Zeitangabe – YYYY-MM-DD HH:MM:SS
* TIMESTAMP Zeitangabe – YYYY-MM-DD HH:MM:SS (berücksichtigt Zeitzonen, gespeichert in UTC)

Bsp.:

CREATE TABLE cellar (bin SMALLINT NOT NULL,

wine VARCHAR(20),

producer VARCHAR(25),

jahr SMALLINT,

bottles SMALLINT,

ready SMALLINT,

comments VARCHAR(30));

Löschen einer Tabelle

DROP TABLE [IF [NOT] EXISTS] Tabellenname;

(Tabelle kopieren)

SELECT \* INTO neuer\_Tabellenname FROM Tabellenname;

Ändern der Tabellenstruktur

Hinzufügen von Spalten:

ALTER TABLE Tabellenname ADD (Spaltenname Datentyp,

Spaltenname Datentyp,…

Spaltenname Datentyp);

Ändern des Datentyps von Spalten:

ALTER TABLE Tabellenname ALTER COLUMN

(Spaltenname Datentyp,…);

Ändern der ganzen Spalte:

ALTER TABLE Tabellenname CHANGE voriger Spaltenname

neuer Spaltenname neuer Datentyp;

Löschen/Entfernen von Spalten aus der Tabelle:

ALTER TABLE Tabellenname DROP COLUMN Spaltenname;

Achtung: Diese Befehle unterscheiden sich teilweise von SQL-Version zu Version! Im Zweifelsfall (im Internet etc.) nachsehen!

Einfügen von Datensätzen

INSERT INTO Tabellenname [(Spaltenname, … , Spaltenname)]

VALUES (Wert, Wert, … , Wert);

Bsp.:

INSERT INTO cellar

VALUES (6, `Chardonnay`, `Chappellet`, 82,

4, 85, `Thanksgiving`);

INSERT INTO cellar (bin, wine, producer, jahr, bottles, ready)

VALUES (11, `Jo.Riesling`, `Buena Vista`,

84, 10, 86);

Kleiner Trick:

INSERT INTO cellar

VALUES (2, `Chardonnay`, `Buena Vista`,

83, 1, 85, ``);

Ändern von Werten

UPDATE Tabellenname SET (Spalte = Ausdruck, … ,

(Spalte = Ausdruck)

[WHERE Bedingung];

Bsp.:

UPDATE cellar SET bottles = 4 WHERE bin = 3;

Löschen von Datensätzen

DELETE FROM Tabellenname [WHERE Bedingung];

Bsp.:

DELETE FROM cellar WHERE bottles = 0;

Einfache „Arithmetik“ in SQL

Bsp.:

UPDATE cellar SET bottles = bottles + 5 WHERE bin = 6;

Bemerkung: „bottles = bottles + 5“, was mathematisch keinen Sinn machen würde, ist in der Computerwelt als Zuweisung zu verstehen: Verwende den ursprünglichen Wert von „bottles“, zähle 5 hinzu und speichere das Ergebnis wiederum in „bottles“!

Obiges Beispiel lässt sich in manchen SQL-„Dialekten“ auch abkürzen:

UPDATE cellar SET bottles += 5 WHERE bin = 6;

Achtung: Dies funktioniert leider nicht in MariaDB/MySQL!

Weitere Bemerkung: Die Ganzzahlige Division lässt sich z.B. in MS-SQL einfach mit „/“ umsetzen; wenn dort Dividend und Divisor beides Integer-Werte sind, so wird ganzzahlig („mit Rest“, Rest wird verworfen) dividiert.

In MariaDB/MySQL muss man zur Ganzzahl-Divison den DIV-Operator verwenden, also z.B.

Bottles = bottles DIV 2;

Anzeigen einer (ganzen) Tabelle

SELECT \* FROM Tabellenname;

Dies „selektiert“ alle („\*“) Datensätze der Tabelle; Ausführlicheres zum SELECT-Befehl folgt noch.

Anfragen (Die SELECT-Anweisung)

Syntax:

SELECT Spaltenname [,…, Spaltenname]

FROM Tabellenname [,…, Tabellenname]

[WHERE Bedingung];

Abkürzende Schreibweise für die Auswahl aller Spalten einer (oder mehrerer) Tabellen:

SELECT \* FROM…

Die WHERE –Bedingung ist ein sog. „Boolescher Ausdruck“ (entweder Wahr oder Falsch), in dem folgende Operatoren vorkommen können (geordnet nach Hierarchie in SQL):

* () Klammern
* ~ Bitwise NOT
* \* (Multiplikation), /(Division), %(Modulus)
* +, - (unär, „Vorzeichen“), + („Concatenation“, Zusammenführen von z.B. Zeichenketten)
* +, - (Addition, Subtraktion), & (Bitwise AND), ^(Bitwise XOR), | (Bitwise OR)
* >=, >, <, <=, <> / !=, = (Vergleichs-Operatoren)
* NOT-Verknüpfung
* **AND,** ALL, ANY, BETWEEN, IN, LIKE, SOME, **OR**
* = (Zuweisung)

Bsp.:

SELECT wine, producer, jahr FROM cellar

WHERE producer = ‚Buena Vista‘ AND bottles > 1;

Ergebnis:

|  |  |  |
| --- | --- | --- |
| Wine | Producer | Jahr |
| Gewürzkraminer | Buena Vista | 80 |

„Beauty-Tip:“

Spaltennamen können beliebig gesetzt werden mit dem „AS“-Schlüsselwort:

SELECT wine AS Weinsorte, producer AS Hersteller…

Sichten (Views, „virtuelle Tabellen“)

Sichten erlauben es, einen individuellen Ausschnitt einer oder mehrerer Tabellen zu erstellen. Eine Sicht ist nicht als Tabelle gespeichert, sondern als „benannte Abfrage“! Die Sicht kann sich ändern, wenn sich Werte in der entsprechenden, zugrundeliegenden Tabelle ändern!

Erstellen einer Sicht:

CREATE VIEW Sichtname

AS SELECT – Anweisung;

Bsp.:

CREATE VIEW bottles\_sicht

AS SELECT bottles, producer, wine

FROM cellar [WHERE Bedingung];

Abfragen von Daten aus einer Sicht:

Bsp.:

SELECT \* FROM bottles\_sicht WHERE bottles >= 5;

Eine Anfrage auf eine Sicht wird vom System in eine Anfrage auf die Basistabelle(n) umgewandelt!

Merke: Das Einfügen, Ändern und Löschen von Datensätzen in einer Sicht ist nur unter Einschränkungen möglich; sie betreffen aber immer die Basistabelle!

Löschen einer Sicht:

DROP VIEW Sichtname;

Geordnetes Selektieren

Bsp.:

SELECT wine, producer, jahr FROM cellar

ORDER BY jahr [DESC | ASC];

Daten werden sortiert nach der ausgewählten Spalte (oder auch Spalten) mit dem Statement „ORDER BY“. Per Default wird aufsteigend sortiert (Zusatz „ASC“); wenn man absteigend sortieren will, muss man das Statement „DESC“ hinzufügen.

Statistische Funktionen

* SUM(Spaltenname) Summe der Spaltenwerte
* AVG(Spaltenname) Durchschnitt der Spaltenwerte
* MAX(Spaltenname) Maximum der Spaltenwerte
* MIN(Spaltenname) Minimum der Spaltenwerte
* COUNT(Spaltenname) Anzahl der Datensätze in der angegebenen Spalte
* COUNT(\*) Anzahl der Datensätze in der ganzen Tabelle/Sicht
* COUNT(DISTINCT Spaltenn.) Anzahl der verschiedenen Datensätze in der Spalte

Bsp.:

SELECT SUM(bottles), COUNT(DISTINCT wine) [weitere, einfache Spalte wie z.B. producer kann hier NICHT eingefügt werden!]

FROM cellar WHERE jahr < 80;

Ergebnis:

|  |  |
| --- | --- |
| (sum) | (count) |
| 34 | 5 |

Falls eine statistische Funktion (Typ: „Aggregats-Funktion“) benutzt wird, müssen auch auf alle anderen Spalten der Abfrage solche Funktionen angewendet werden!

(Ausnahme: Es wird nach den ausgewählten Spalten gruppiert! (siehe unten!))

Gruppierung

Statt einzelner Datensätze werden Gruppen von Datensätzen gebildet, die in bestimmten Spalten (angegeben durch das „GROUP BY“-Statement) denselben Wert haben!

Bsp.:

SELECT producer, SUM(bottles) FROM cellar

GROUP BY producer;

Ergibt:

|  |  |
| --- | --- |
| Producer | (sum) |
| Buena Vista | 5 |
| Louis Martini | 5 |
| Ch.St.Jean | 6 |
| … | … |

Die statistischen Funktionen (Hier: „SUM()“) beziehen sich in diesem Zusammenhang jeweils auf eine Gruppe!

Ähnlich wie zuvor dürfen in der SELECT-Anweisung nur „GROUP BY“-Spalten und statistische (zusammenfassende) Funktionen vorkommen!

Auswahl von Daten in Gruppen:

Bsp.:

SELECT producer, SUM(bottles) FROM cellar

GROUP BY producer

HAVING COUNT(\*) > 1;

Ergibt:

|  |  |
| --- | --- |
| Producer | (sum) |
| Buena Vista | 5 |
| Ch.St.Jean | 6 |
| Mirassou | 18 |
| Robt.Mondavi | 16 |

Die „HAVING“-Klausel für Gruppen entspricht der „WHERE“-Klausel für normale Datensätze!

Komplexe Anfragen – Joins

Beispieltabellen:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tabelle R | |  | Tabelle S | |
| A | B | B | C |
| a | 0 | 1 | c |
| b | 1 | 0 | d |

Kartesisches Produkt („Join“)

SELECT \* FROM R, S;

Ergibt:

|  |  |  |  |
| --- | --- | --- | --- |
|  | R.B | S.B |  |
| A | B | B | C |
| a | 0 | 1 | c |
| a | 0 | 0 | d |
| b | 1 | 1 | c |
| b | 1 | 0 | d |

Wenn R n Zeilen und S m Zeilen hat, dann hat das „kartesische Produkt“ aus R und S n \* m Zeilen!

Equi-Join:

Es werden nur Zeilen von R und S kombiniert, die in der Spalte B denselben Wert haben

SELECT \* FROM R, S WHERE R.B = S.B;

Ergebnis:

|  |  |  |  |
| --- | --- | --- | --- |
|  | R.B | S.B |  |
| A | B | B | C |
| a | 0 | 0 | d |
| b | 1 | 1 | c |

Natürlicher Join („Natural Join“)

Wie Equi-Join, wobei doppelte Spalten nur einmal angezeigt werden.

SELECT A, R.B, C FROM R, S WHERE R.B = S.B;

(Oder: SELECT R.\*, C FROM …, sowie die umgekehrten Versionen)

Liefert:

|  |  |  |
| --- | --- | --- |
| A | B | C |
| a | 0 | d |
| b | 1 | c |

Ausführliche Beispiele zu den Joins -> Übungen!

* Join-Operationen führen also Datensätze aus zwei oder mehreren Tabellen zusammen, wenn die angegebenen WHERE-Kriterien erfüllt werden. Sind eine oder mehrere Kriterien in der WHERE-Klausel nicht erfüllt, so entsteht kein Datensatz in der Ergebnistabelle!
* Umgekehrt gilt: Ist kein Kriterium (WHERE-Bedingung) vorhanden, wird „alles mit allem“ verknüpft -> auch bekannt als „kartesisches Produkt“ in der Mathematik.
* Im Beispiel oben musste der Spaltenname für die Spalte „B“ gegebenenfalls mit der Herkunftstabelle zusammen angegeben werden in der Form „Tabellenname.Spaltenname“.  
  Dies ist nötig bei Spaltennamen, die in der Abfrage mehrfach vorkommen; bei „eindeutigen“ Spaltennamen kann man auch den Tabellennamen weglassen!

LEFT/RIGHT JOIN (und weiteres)

* Die Beispiele im „Joins-Abschnitt stellen in der SQL-Syntax den Spezialfall eines sogenannten „INNER JOINS“ dar! Eine alternative Syntax zu den obigen Beispielen wäre:

SELECT A, R.B, C FROM R INNER JOIN S ON R.B = S.B;

* Es gibt nun an der Stelle, wo oben „INNER JOIN“ steht, mehrere Optionen in der Syntax, die verschiedene Ausgaben erzeugen:
* LEFT JOIN: Ein Datensatz aus der „linken“ (erstgenannten) Tabelle kommt in jedem Fall ins Ergebnis. Die Daten betreffend der „rechten“ (zweiten) Tabelle werden nur eingetragen, wenn das „ON“-Kriterium (vorher: „WHERE“) erfüllt ist; ansonsten bleiben die Spalten leer (NULL-Werte)!
* RIGHT JOIN: Hier gilt der umgekehrte Fall: Daten der „linken“ (erstgenannten) Tabelle bleiben leer, wenn das ON-Kriterium nicht erfüllt ist. Die Daten der „rechten“ (zweiten) Tabelle werden immer eingetragen!
* Beachte: Im Gegensatz zu den vorigen Beispielen spielt die Reihenfolge der genannten Tabellen hier natürlich die entscheidende Rolle!

Bsp.: In einer Rechnungs-Datenbank sollen alle Rechnungen gesucht werden. Falls per Kreditkarte bezahlt wurde, sollen auch Kartendaten ausgegeben werden!

SELECT RechnungsNr, KundenNr, Betrag, Rechnungen.KartenNr, Kartendaten.Firma, Inhaber, AblDatum FROM Rechnungen

LEFT JOIN Kartendaten ON Kartendaten.KartenNr = Rechnungen.KartenNr;

Ausgabe z.B.:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| RechnungsNr | KundenNr | Betrag | Rechnungen.KartenNr | Firma | Inhaber | AblDatum |
| 58765 | ABC123 | 49,95 | 12345 | VISA | M.Maier | 03/21 |
| 58766 | ABC234 | 12,95 |  |  |  |  |
| 58767 | ABC123 | 79,95 | 12345 | VISA | M.Maier | 03/21 |

Achtung: Hier spielt die Herkunft der selektierten Kartennummer eine Rolle! (Bei Verwendung der „rechten“ Kartennummer werden eventuell Rechnungen, die mit ein und derselben Kreditkarte bezahlt wurden, nicht korrekt angezeigt!)

Die verschiedenen Verknüpfungs-Arten, in Diagrammen visualisiert:





* (INNER) JOIN: Nur die „Schnittmenge“ der beiden Tabellen wird geliefert; Daten werden nur aufgenommen, wo es passende Werte in beiden Tabellen gibt.
* LEFT (OUTER) JOIN: Die erste Tabelle wird komplett aufgenommen; von der zweiten Tabelle werden die Daten hinzugefügt, wo es passende Werte in beiden Tabellen gibt.
* RIGHT (OUTER) JOIN: Umgekehrt; Die zweite Tabelle wird komplett aufgenommen; von der ersten Tabelle werden Daten hinzugefügt, wo es passende Werte gibt.
* FULL (OUTER) JOIN: Alle Daten werden aufgenommen; bei keiner Übereinstimmung zwischen den Tabellen bleiben Datenfelder leer.

Die Statements „INNER“ und „OUTER geben zusätzlich noch einmal das Einschlussverhalten an und können meist weggelassen werden!

Geschachtelte Anfragen (Unteranfragen)

Unterabfragen erlauben es, die Auswahl von Daten in einer Tabelle in Abhängigkeit von Werten anderer Tabellen zu treffen. Dazu können die Operatoren IN oder NOT IN verwendet werden.

Bsp.:

|  |  |
| --- | --- |
| Tabelle Angestellte | |
| Name | Proj.Nr |
| Müller | 1 |
| Maier | 2 |

|  |  |
| --- | --- |
| Tabelle Projekte | |
| Proj.Nr | Projektname |
| 1 | Gehalt |
| 2 | Ausgaben |

SELECT Name FROM Angestellte

WHERE Proj.Nr IN

(SELECT Proj.Nr FROM Projekt

WHERE Projektname = ‚Ausgaben‘);

Die „innere“ Abfrage (innerhalb der Klammern) liefert zunächst zurück: Proj.Nr / 2

Also hieße die äußere Abfrage hier:

SELECT Name FROM Angestellte

WHERE Proj.Nr = 2;

SQL Injection Attacks und Schutz davor

Achtung: Hacking ist strafbar ohne Erlaubnis des Eigentümers der jeweiligen Website. Trotzdem soll die Strategie dahinter kurz betrachtet werden, um sich davor schützen zu können!

Bsp. In „Pseudocode“:

TxtUserID = getRequestString(„UserID“); 🡨 vom User eingegebene Daten; evtl. über ein Textfeld

txtSQL = „SELECT \* FROM Users WHERE UserID = ‚“ + TxtUserID + „‘;“;

„txtSQL“ wird dann als Query an die DB geschickt (über PHP/MySQL)

Ein User sieht dann auf der Webseite ein Eingabefeld, das zum Eingeben von „UserID“ auffordert. Solange der User genau das tut, also eine ID eingibt (z.B. 99 oder 105 oder… ), dann ist alles in Ordnung.

Bei etwas böserer Absicht kann der User auch eingeben:

105‘ OR 1=1‘

* SELECT \* FROM Users WHERE UserID = ‚105‘ OR 1=1‘‘; 🡪 immer wahr!
* Ausgabe der gesamten User-Liste! Gefährlich, weil evtl. sensitive Daten aufgedeckt werden können!

Mit noch mehr böser Absicht kann der User eingeben:

105‘; DROP TABLE Users

* SELECT \* FROM Users WHERE UserID = ‚105‘; DROP TABLE USERS;
* User-Tabelle wird unwiederbringlich gelöscht!

Viel Hacking-Potential mit SQL-Injections.

Abhilfe: User-Eingaben müssen strikt als Parameter und nicht als (Teil eines) Befehles interpretiert werden! Dazu gibt es verschiedene Lösungen in verschiedenen Sprachen.

Bsp. In PHP (am besten aber selbst nach den neuesten Methoden suchen!):

$stmt = $dbh->prepare(„INSERT INTO customers

(CustomerName, Address, City) VALUES (:nam, :add, :cit)“);

$stmt->bindParam(‚:nam‘, $txtNam);

$stmt->bindParam(‚:add‘, $txtAdd);

$stmt->bindParam(‚:cit‘, $txtCit);

* „bereinigte“ SQL-Abfrage; Exploits können so (weitestgehend) vermieden werden.